# Discrete event modeling: VHDL

Peter Marwedel
Informatik 12
Univ. Dortmund
Germany

# Models of computation

VHDL as a prominent example of discrete event modeling:

| ***Communication/ Computation*** | **Shared memory** | **Message passing** | |
|---|---|---|---|
| | | **blocking** | **Non-blocking** |
| FSM | StateCharts | | SDL |
| Data flow model | | | Kahn process networks, SDL |
| Imperative von Neumann computing | C, C++, Java | C, C++, Java with message passing libraries | |
| | | CSP, ADA | |
| Discrete event model | VHDL, Verilog, SystemC | Just experimental systems, e.g. distributed discrete event simulation in Ptolemy | |

# VHDL

HDL = hardware description language

Textual HDLs replaced graphical HDLs in the 1980‘ies (better description of complex behavior).

In this course:

VHDL = VHSIC hardware description language

VHSIC = very high speed integrated circuit

1980: Def. started by US Dept. of Defense (DoD) in 1980

1984: first version of the language defined, based on ADA (which in turn is based on PASCAL)

1987: revised version became IEEE standard 1076

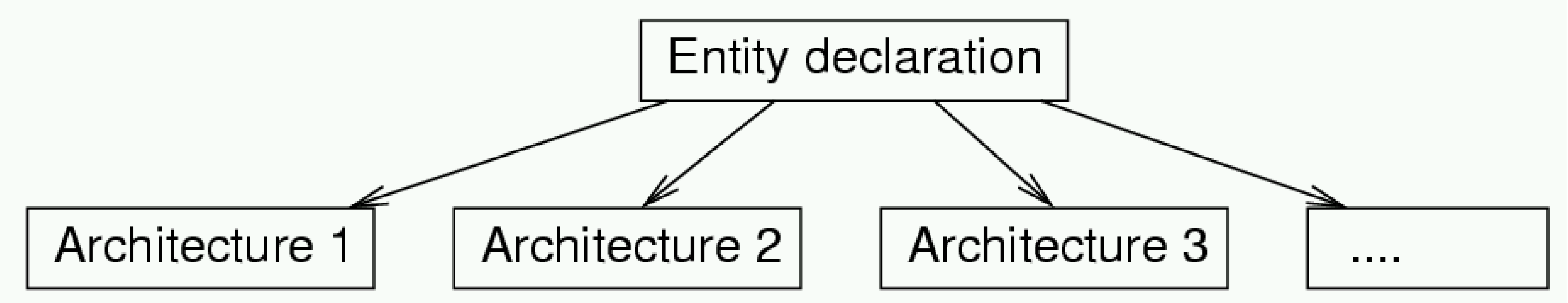
1992: revised IEEE standard

1999: VHDL-AMS: includes analog modeling

2006: Major extensions

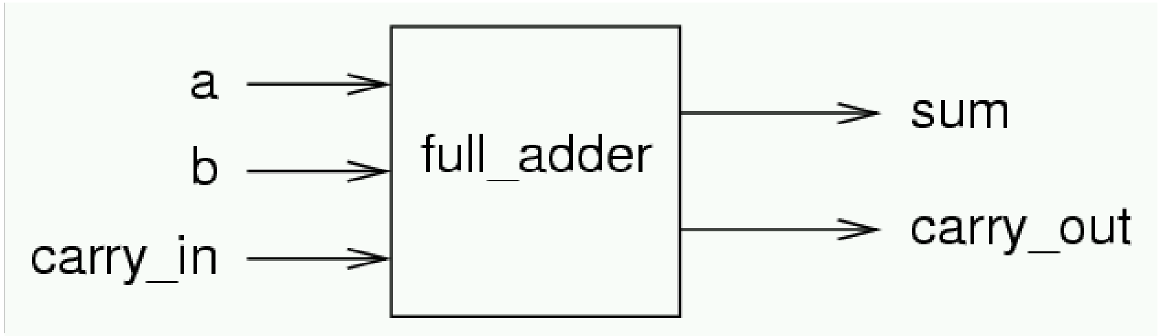
# Entities and architectures

Each design unit is called an **entity**.
Entities are comprised of **entity declarations** and one or several **architectures.**




Each architecture includes a model of the entity. By default, the most recently analyzed architecture is used. The use of another architecture can be requested in a **configuration**.

# The full adder as an example
# - Entity declaration -




## Entity declaration:

**entity** full_adder **is**
 **port**(a, b, carry_in: **in** Bit;  -- input ports
  sum,carry_out: **out** Bit); --output ports
**end** full_adder;

# The full adder as an example - Architectures -

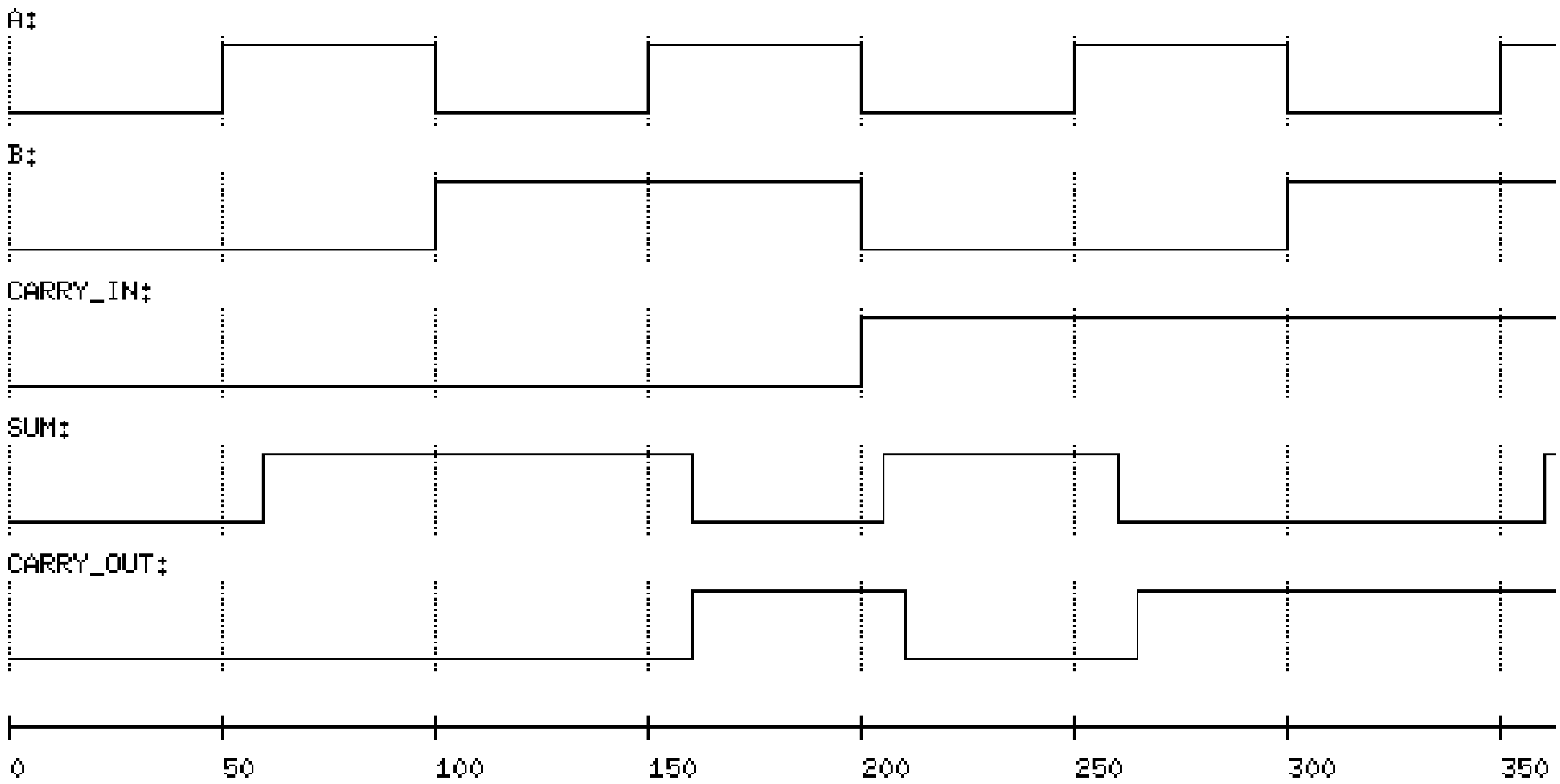
Architecture = Architecture header + architectural bodies

```
architecture behavior of full_adder is
 begin
  sum       <= (a xor b) xor carry_in after 10 Ns;
  carry_out <= (a and b) or (a and carry_in) or
               (b and carry_in)           after 10 Ns;
 end behavior;
```

Architectural bodies can be
- **behavioral bodies** or - **structural bodies**.

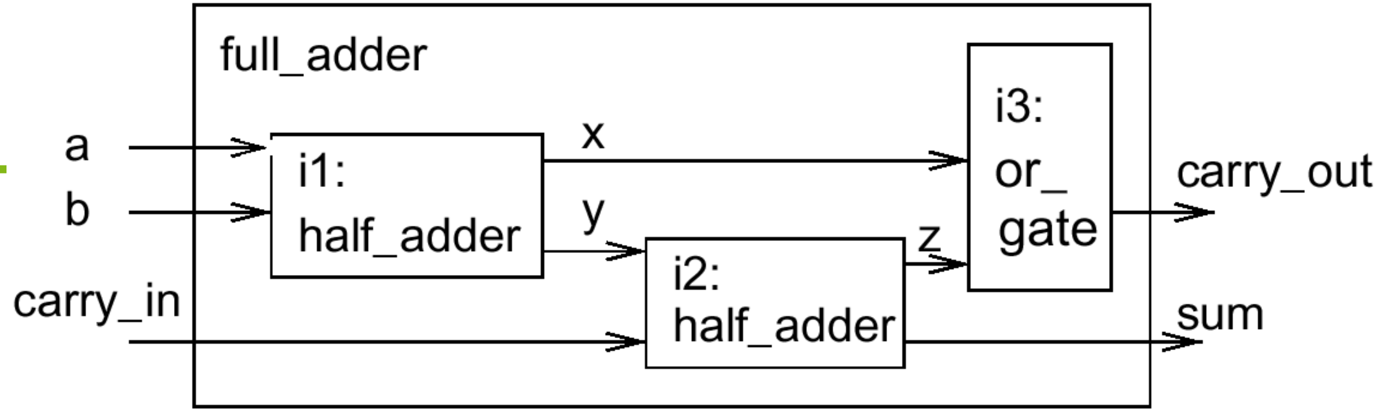
Bodies not referring to hardware components are called behavioral bodies.

# The full adder as an example
# - Simulation results -




**Behavioral description different from the one shown (includes 5ns delays).**

# Structural bodies




```
architecture structure of full_adder is
  component half_adder
    port (in1,in2:in Bit; carry:out Bit; sum:out Bit);
  end component;
  component or_gate
    port (in1, in2:in Bit; o:out Bit);
  end component;
 signal x, y, z: Bit;       -- local signals
 begin                      -- port map section
   i1: half_adder port map (a, b, x, y);
   i2: half_adder port map (y, carry_in, z, sum);
   i3: or_gate    port map (x, z, carry_out);
 end structure;
```

# Multi-valued logic and standard IEEE 1164

How many logic values for modeling?
Two ('0' and '1') or more?
If real circuits have to be described, some abstraction of the resistance (inversely-related to the strength) is required.
☞ We introduce the distinction between:
- the **logic level** (as an abstraction of the voltage) and
- the **strength** (as an abstraction of the current drive capability) of a signal.

The two are encoded in **logic values**.
☞ CSA (connector, switch, attenuator) - theory [Hayes]

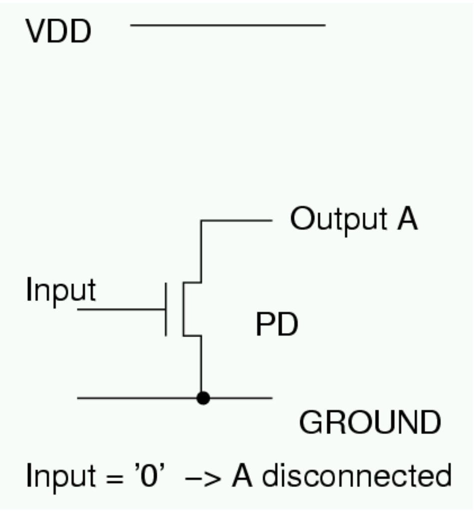
# 1 signal strength

Logic values '0' and '1'.
Both of the same strength.
Encoding false and true, respectively.

# 2 signal strengths

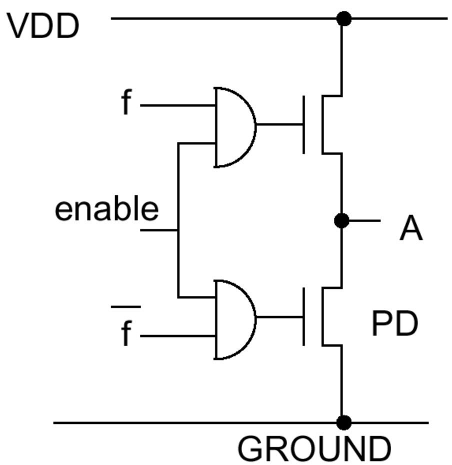
Many subcircuits can effectively disconnect themselves from the rest of the circuit (they provide „high impedance“ values to the rest of the circuit). Example: subcircuits with open collector or tri-state outputs.

Open-collector



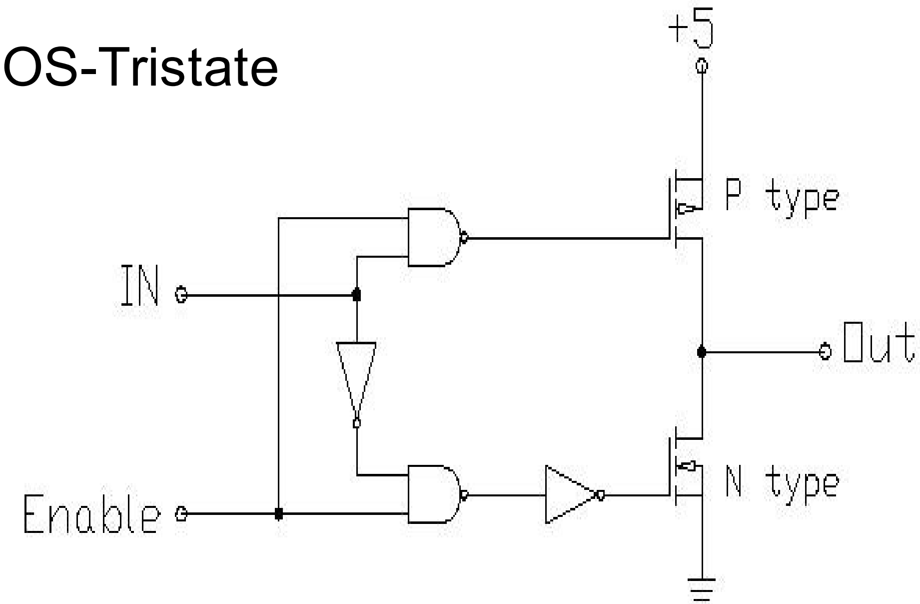
# TriState circuits

nMOS-Tristate




enable = ’0’ –> A disconnected

CMOS-Tristate

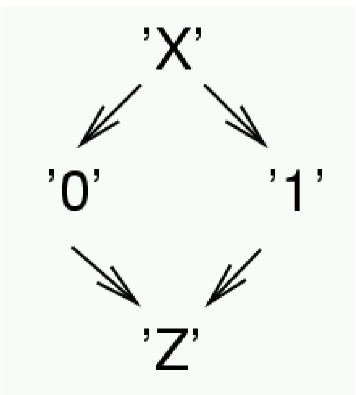



Source: http://www-unix.oit.umass.edu/~phys532/lecture3.pdf

☞ We introduce signal value 'Z', meaning „high impedance “

# 2 signal strengths (cont'ed)

We introduce an operation #, which generates the effective signal value whenever two signals are connected by a wire. #('0','Z')='0'; #('1','Z')='1'; '0' and '1' are „stronger" than 'Z'

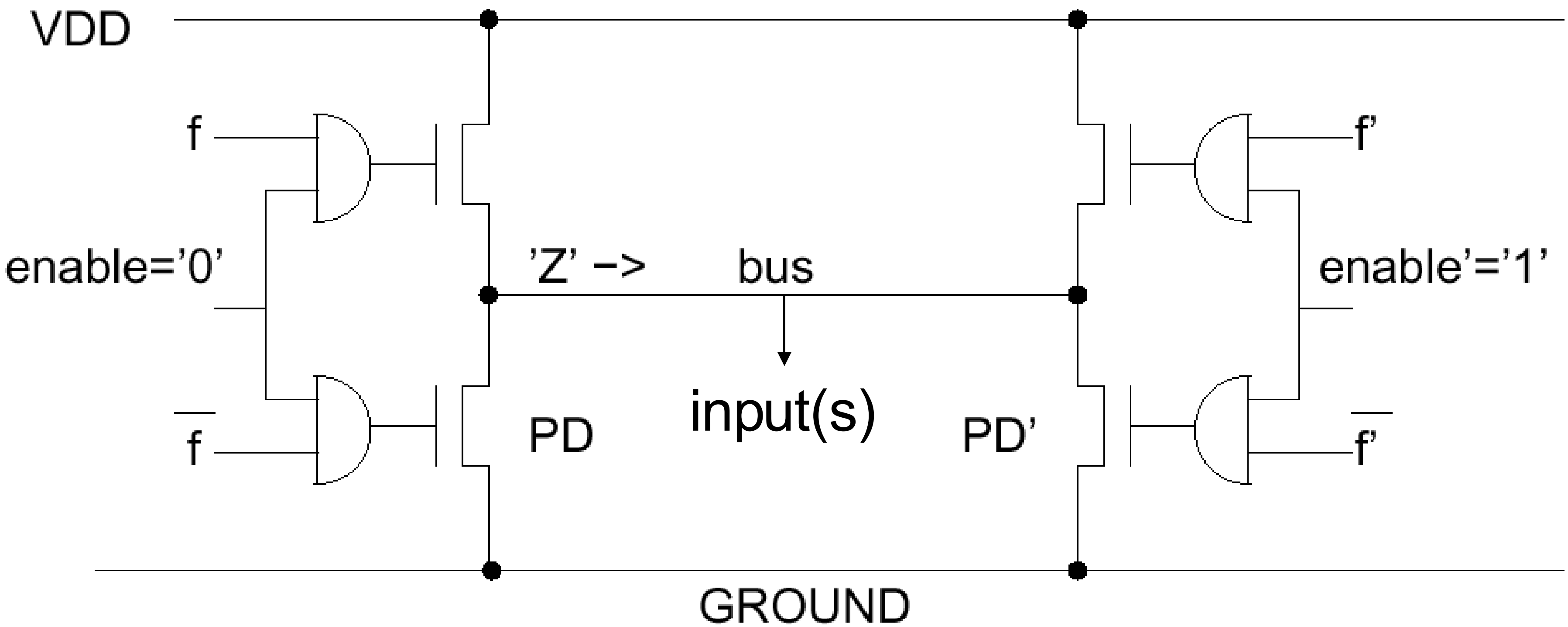



1 strength

According to the partial order in the diagram, # returns the *larger of the two arguments*.

In order to define #('0','1'), we introduce 'X', denoting an undefined signal level. 'X' has the same strength as '0' and '1'.

# Application example

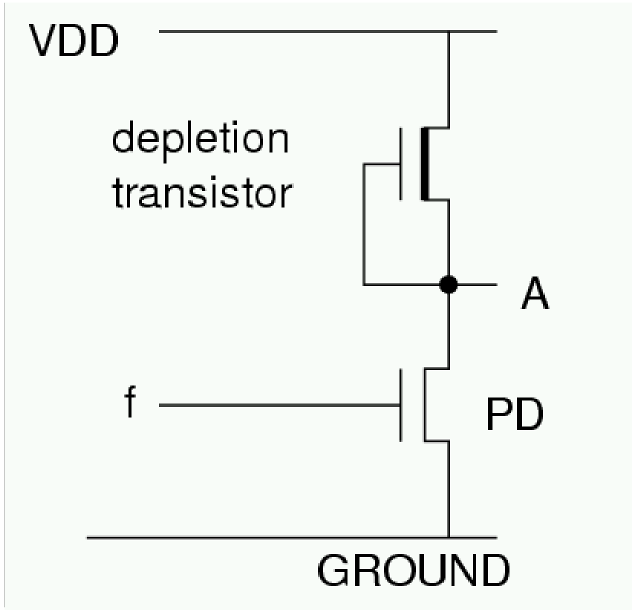



signal value on bus = #(value from left subcircuit, value from right subcircuit)
#('Z', value from right subcircuit)
value from right subcircuit

„as if left circuit were not there".

# 3 signal strengths

Current values insufficient for describing real circuits:




Depletion transistor contributes a weak value to be considered in the #-operation for signal A

☞ Introduction of 'H', denoting a weak signal of the same level as '1'.

#('H', '0')='0'; #('H','Z') = 'H'

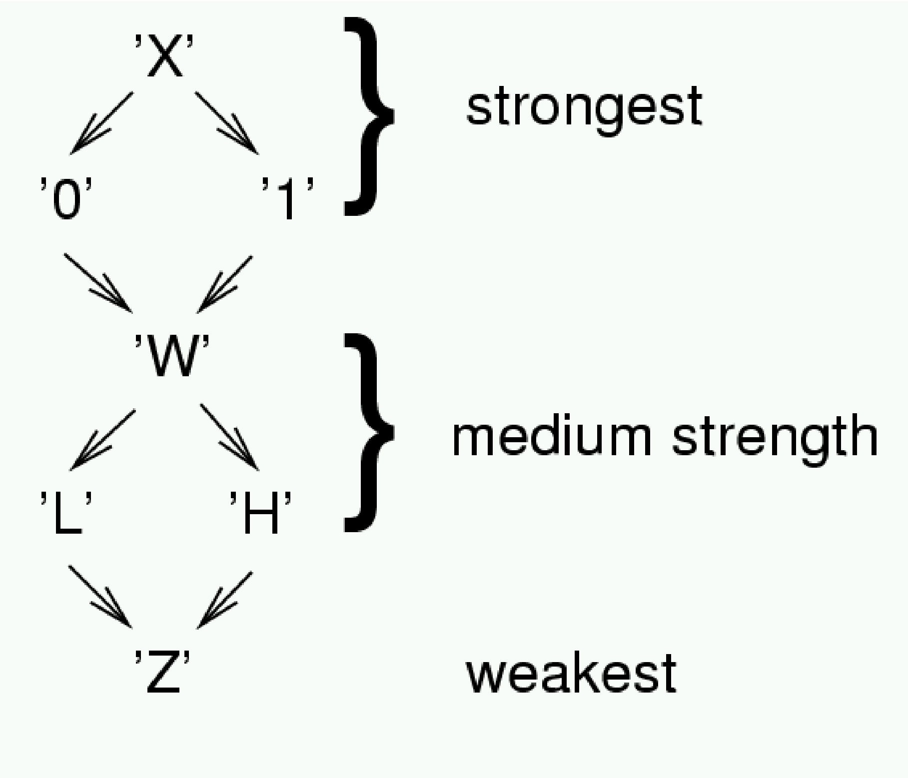
# 3 signal strengths

There may also be weak signals of the same level as '0'

☞ Introduction of 'L', denoting a weak signal of the same level as '0': #('L', '0')='0'; #('L,'Z') = 'L';

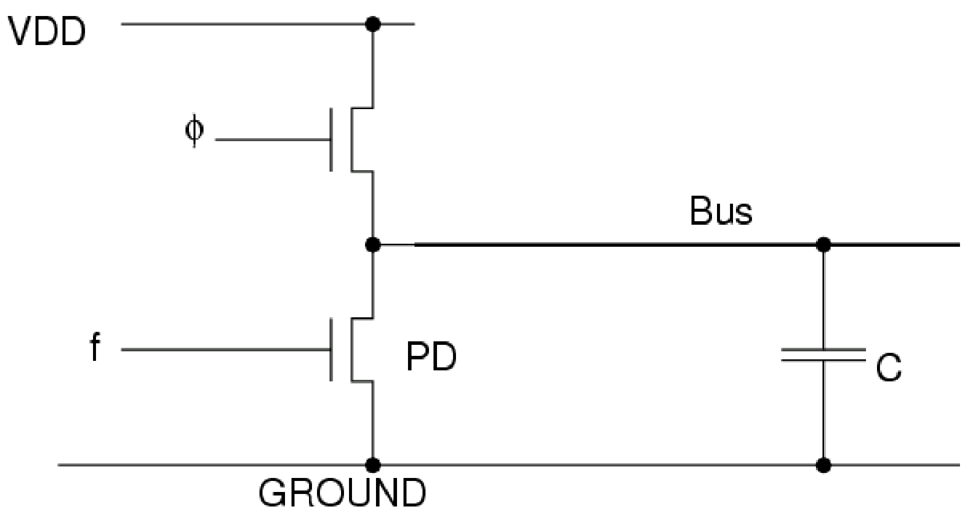
☞ Introduction of 'W', denoting a weak signal of the same level as 'X': #('L', 'H')='W'; #('L,'W') = 'W';

# reflected by the partial order shown.



# 4 signal strengths (1)

Current values insufficient for describing pre-charging:




Pre-charged '1'-levels weaker than any of the values considered so far, except 'Z'.

☞ Introduction of 'h', denoting a very weak signal of the same level as '1'.

#('h', '0')='0'; #('h','Z') = 'h'

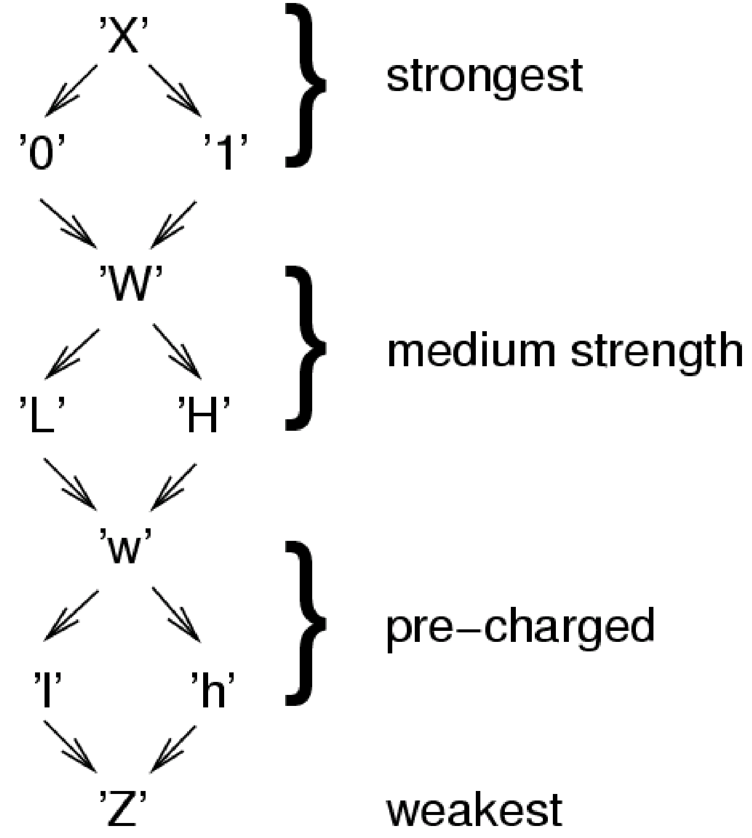
# 4 signal strengths (2)

There may also be weak signals of the same level as '0'

☞ Introduction of 'l', denoting a very weak signal of the same level as '0': #('l', '0')='0'; #('l,'Z') = 'l';

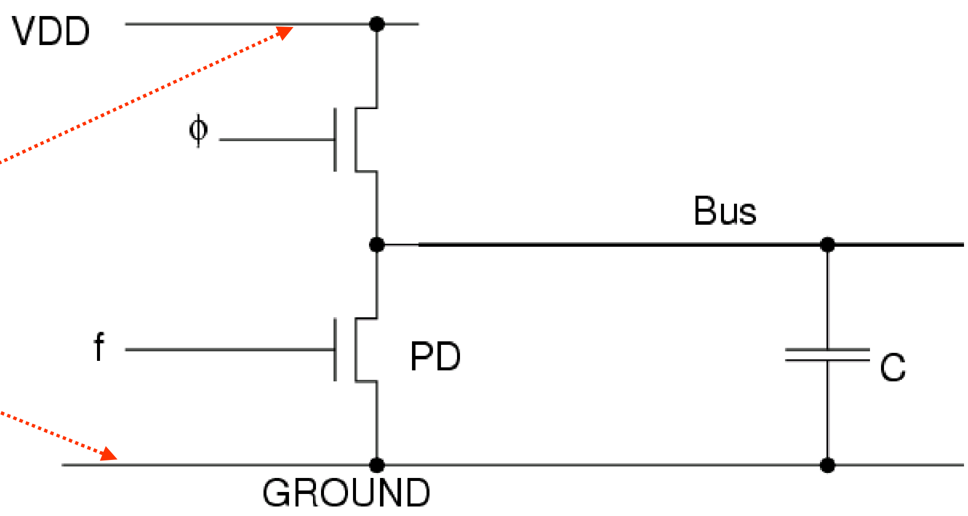
☞ Introduction of 'w', denoting a very weak signal of the same level as 'W': #('l', 'h')='w'; #('h','w') = 'w'; ...

# reflected by the partial order shown.



# 5 signal strengths

Current values insufficient for describing strength of supply voltage




Supply voltage stronger than any voltage considered so far.

☞ Introduction of 'F0' and 'F1', denoting a very strong signal of the same level as '0 ' and '1'.

☞ Definition of 46-valued logic, also modeling uncertainty (Coelho); initially popular, now hardly used.

# IEEE 1164

VHDL allows user-defined value sets.

☞ Each model could use different value sets (unpractical)

☞ Definition of standard value set according to standard IEEE 1164:

{'0', '1', 'Z', 'X', 'H', 'L', 'W', 'U', '-'}

First seven values as discussed previously.

☞: Everything said about 7-valued logic applies.

☞: Combination of pre-charging and depletion transistors cannot be described in IEEE 1164.

'U': un-initialized signal; used by simulator to initialize all not explicitly initialized signals.

# Input don‘t care

`'-'` denotes **input don't care.**

Suppose:

$f(a,b,c) = a\overline{b} + bc$ except for *a*=*b*=*c*='0' where *f* is undefined

Then, we could like specifying this in VHDL as

f <= **select** a & b & c

    '1' **when** "10-"    -- first term

    '1' **when** "-11"    -- second term

    'X' **when** "000"   -- 'X' ≙ ('0' or '1') here (output don't care)

    '0' **otherwise;**

Simulator would check if  a & b & c = "10-", i.e. if c='-'.

Since c is never assigned a value of '-', this test would always fail. Simulator does not know that `'-'` means either `'1'` or `'0'`, since it does not include any special handling for `'-'`,

(*at least not for pre-VHDL’2006*).

# Function std_match

Special meaning of `'-'` can be used in special function std_match.

**if** std_match(a&b&c,"10-")
is true for any value of **c**, but this does not enable the use of the compact **select** statement.

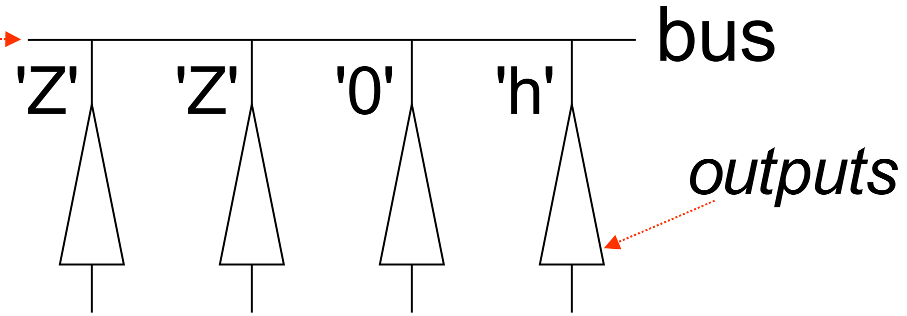
- ☞ The flexibility of VHDL comes at the price of less convenient specifications of Boolean functions.

VHDL'2006 has changed this: `'-'` can be used in the "intended" way in case selectors

# Outputs tied together

In hardware, connected outputs can be used:

Resolution function used for assignments to bus, if bus is declared as std_logic.




Modeling in VHDL: resolution functions

**type** std_ulogic **is** ('U', 'X','0', '1', 'Z', 'W', 'l', 'h', '-');

**subtype** std_logic **is** resolved std_ulogic;

-- involve function resolved for assignments to std_logic

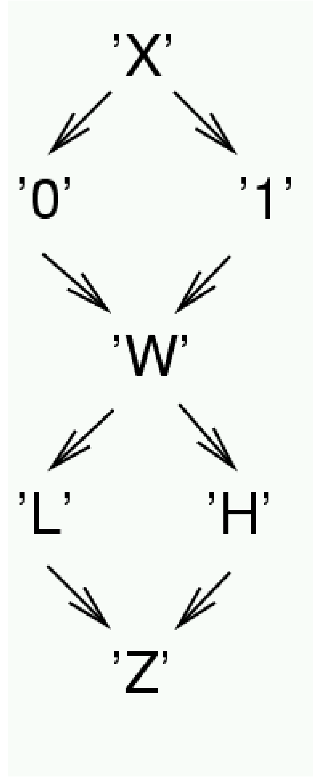
# Resolution function for IEEE 1164

```
type std_ulogic_vector is array(natural range<>)of std_ulogic;

function resolved (s:std_ulogic_vector) return std_ulogic is
  variable result: std_ulogic:='Z';   --weakest value is default
  begin
    if (s'length=1) then return s(s'low) --no resolution
    else for i in s'range loop
      result:=resolution_table(result,s(i))
    end loop
    end if;
   return result;
  end resolved;
```

# Using # (=sup) in resolution functions

```
constant resolution_table : stdlogic_table := (
--U    X    0    1    Z   W     L    H   –
('U', 'U', 'U', 'U', 'U', 'U',  'U', 'U', 'U'),  --| U |
('U', 'X', 'X', 'X', 'X', 'X',  'X', 'X', 'X'),  --| X |
('U', 'X', '0', 'X', '0', '0',  '0', '0', 'X'),  --| 0 |
('U', 'X', 'X', '1', '1', '1',  '1', '1', 'X'),  --| 1 |
('U', 'X', '0', '1', 'Z', 'W',  'L', 'H', 'X'),  --| Z |
('U', 'X', '0', '1', 'W', 'W',  'W', 'H', 'X'),  --| W |
('U', 'X', '0', '1', 'L', 'W',  'L', 'W', 'X'),  --| L |
('U', 'X', '0', '1', 'H', 'W',  'W', 'H', 'X'),  --| H |
('U', 'X', 'X', 'X', 'X', 'X',  'X', 'X', 'X')   --| - |
);
```




This table would be difficult to understand without the partial order

# VHDL processes

**Processes model parallelism in hardware**

General syntax:

*label:* --optional

**process**

  *declarations* --optional

**begin**

  *statements* --optional

**end process**

a <= b **after** 10 ns is equivalent to

**process**

 **begin**

  a <= b **after** 10 ns

 **end**

# Assignments

2 kinds of assignments:

- **<u>Variable assignments</u>**
  **Syntax:** *variable := expression;*
- **<u>Signal assignments</u>**
  **Syntax:**
  *signal* <= *expression*;
  *signal* <= *expression* **after** *delay*;
  *signal* <= **transport** expression **after** delay;
  *signal* <= **reject** time **inertial** expression **after** delay;

Possibly several assignments to 1 signal within 1 process.

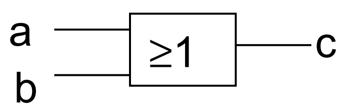
For each signal there is one **driver** per process.
Driver stores information about the **future** of signal,
the so-called **projected waveform.**

# Transport delay

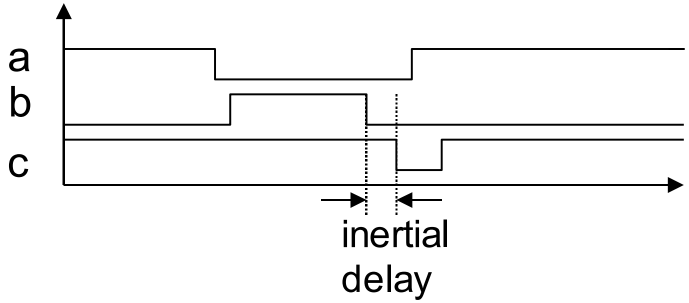
- For transport delay assignments, queued events are never removed again.
- Pulses, no matter how short, will be propagated.
- This corresponds to models for simple wires

# Inertial delay

- If the keyword "transport" is not used, inertial delay is assumed, except after executing the assignment.
- The goal of inertial delay is to suppress all signal "spikes", which are shorter than the delay, resp. shorter than the indicated suppression threshold.
- Inertial delay models the behavior of gates.



OR gate




- Precise rules for when to remove events from the projected waveform are tricky

# Wait-statements

Four possible kinds of **wait**-statements:

- **wait on** *signal list*;
  - wait until signal changes;
  - Example: **wait on** a;
- **wait until** *condition*;
  - wait until condition is met;
  - Example: **wait until** c='1';
- **wait for** *duration*;
  - wait for specified amount of time;
  - Example: **wait for** 10 ns;
- **wait;**
  - suspend indefinitely

# Sensivity lists

Sensivity lists are a shorthand for a single **wait on**-statement at the end of the process body:

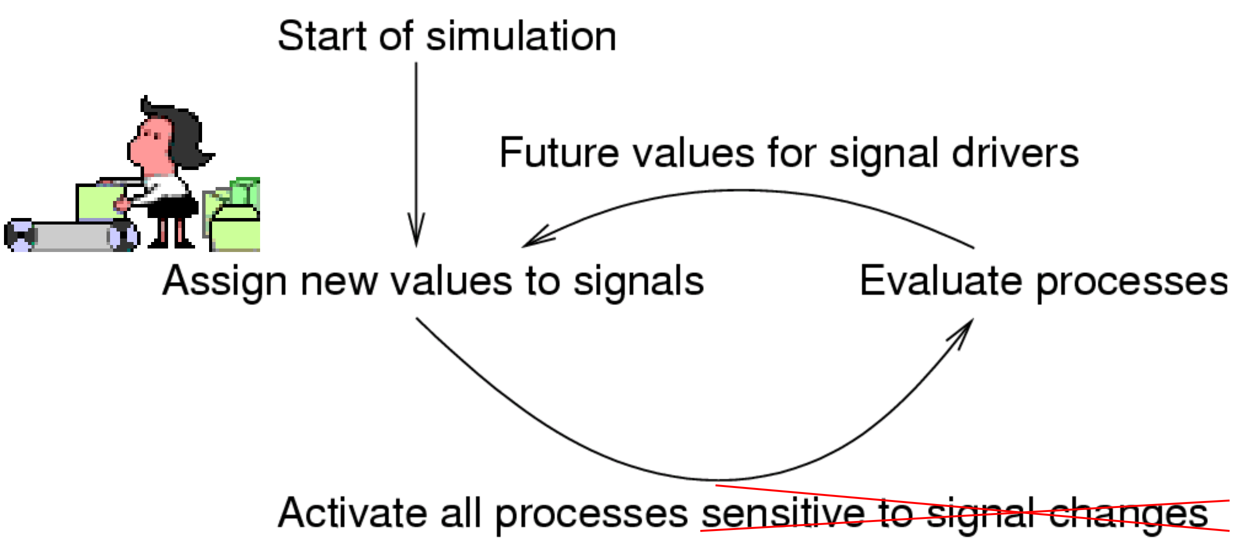
**process** (x, y)
 **begin**
  prod <= x  and y ;
 **end process**;

is equivalent to

**process**
 **begin**
  prod <= x  and y ;
  **wait on** x,y;
 **end process**;

# VHDL semantics: global control

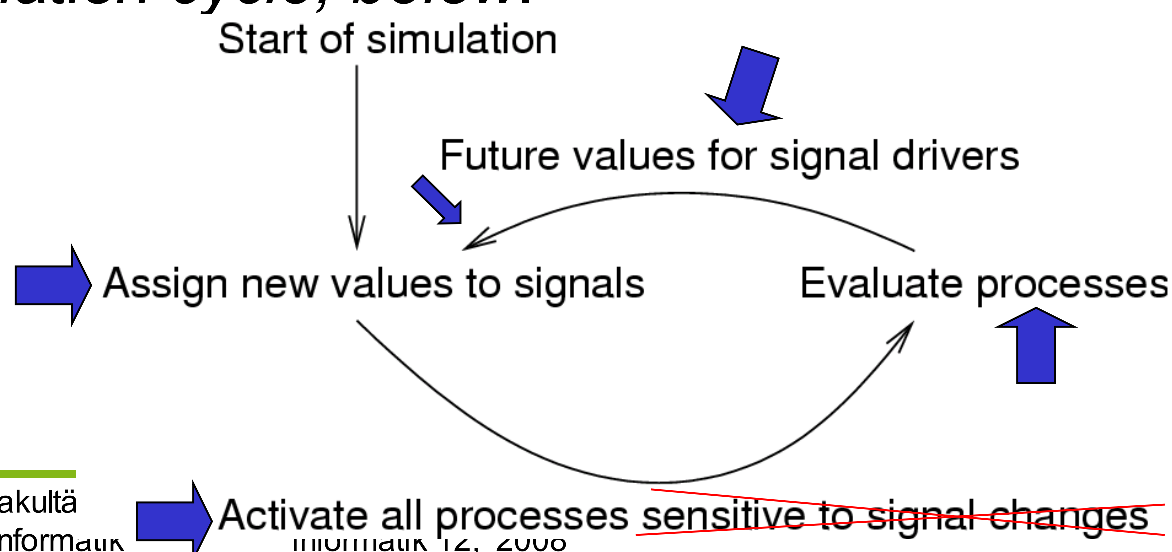
According to the original standards document:
The execution of a model consists of an initialization phase followed by the repetitive execution of process statements in the description of that model.
Initialization phase executes each process once.



# VHDL semantics: initialization

At the beginning of initialization, the current time, $T_c$ is 0 ns.

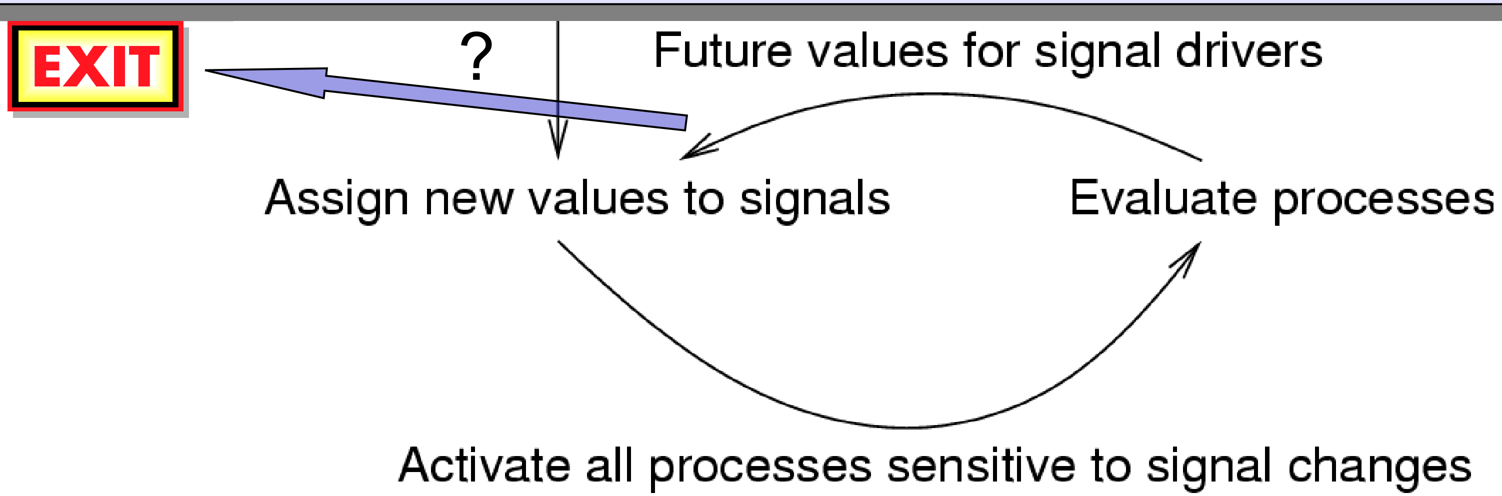
- *The driving value and the effective value of each explicitly declared signal are computed, and the current value of the signal is set to the effective value. …*
- *Each ... process … is executed until it suspends*.
- *The time of the next simulation cycle (… in this case … the 1st cycle), $T_n$ is calculated according to the rules of step f of the simulation cycle, below*.



# VHDL semantics: The simulation cycle (1)

Each simulation cycle starts with setting $T_c$ to $T_n$. $T_n$ was either computed during the initialization or during the last execution of the simulation cycle. Simulation terminates when the current time reaches its maximum, TIME'HIGH. According to the standard, the simulation cycle is as follows:

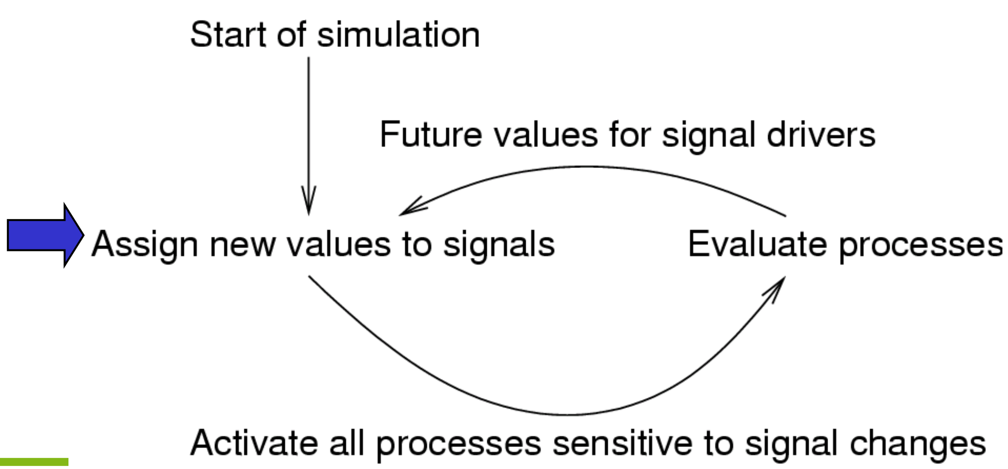
a) The current time, $T_c$ is set to $T_n$. Stop if $T_n$= TIME'HIGH and not ∃ active drivers or process resumptions at $T_n$.



# VHDL semantics: The simulation cycle (2)

a) ***Each active explicit signal in the model is updated. (Events may occur as a result.)***
   Previously computed entries in the queue are now assigned if their time corresponds to the current time $T_c$.
   New values of signals are not assigned before the next simulation cycle, at the earliest.
   Signal value changes result in events ☞ enable the execution of processes that are sensitive to that signal.

b) ..

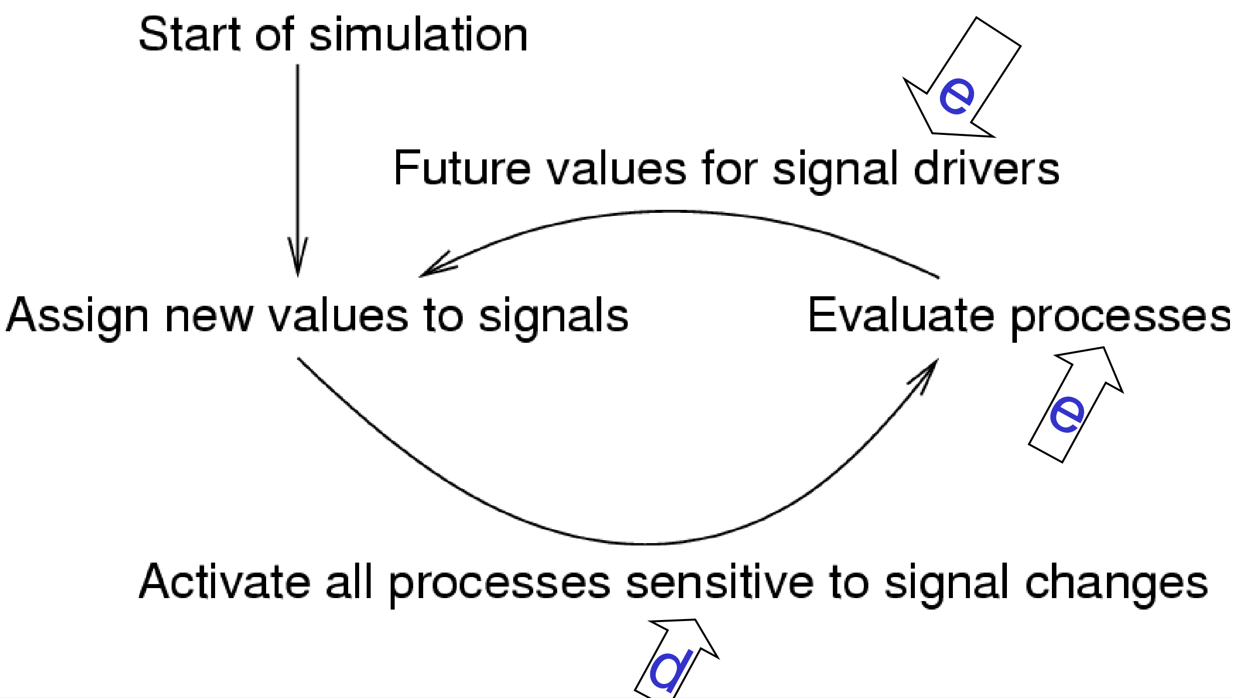


# VHDL semantics: The simulation cycle (3)

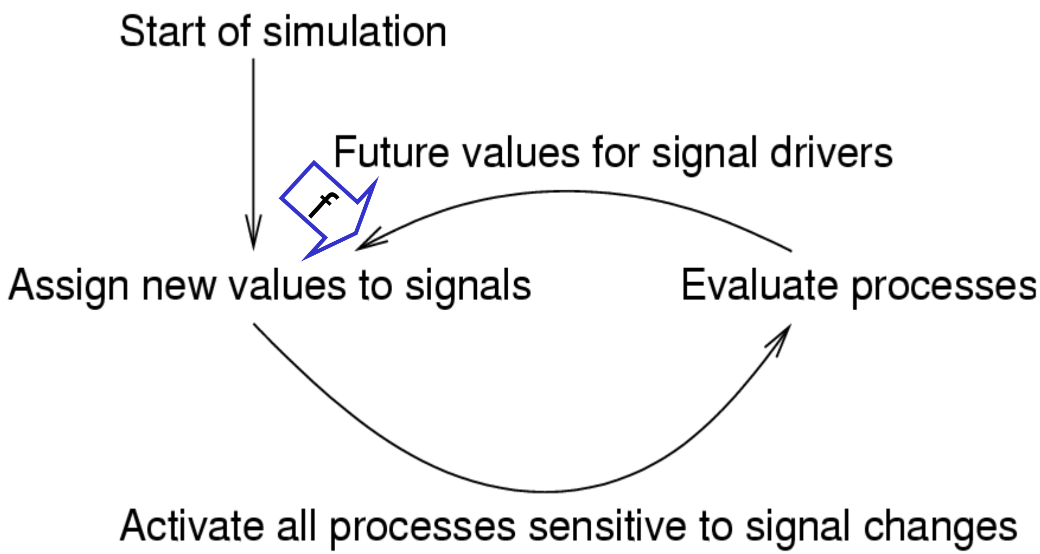
α) ∀ P sensitive to s: if event on s in current cycle: P resumes.

b) Each ... process that has resumed in the current simulation cycle is executed until it suspends*.

*Generates future values for signal drivers.



# VHDL semantics: The simulation cycle (4)




a) Time $T_n$ of the next simulation cycle = earliest of

1. TIME'HIGH (end of simulation time).
2. The next time at which a driver becomes active
3. The next time at which a process resumes (determined by **wait for** statements).

Next simulation cycle (if any) will be a delta cycle if $T_n = T_c$.

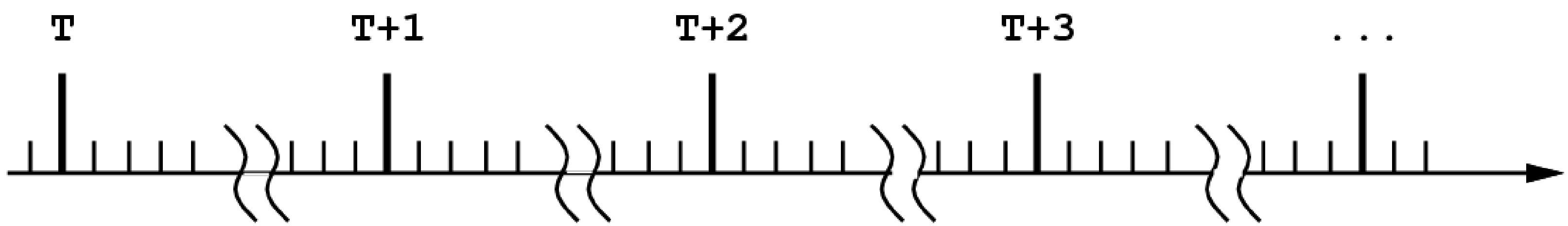
# δ-simulation cycles

…

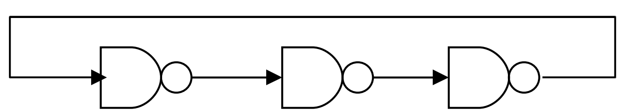
Next simulation cycle (if any) will be a delta cycle if $T_n = T_c$.

Delta cycles are generated for delay-less models.

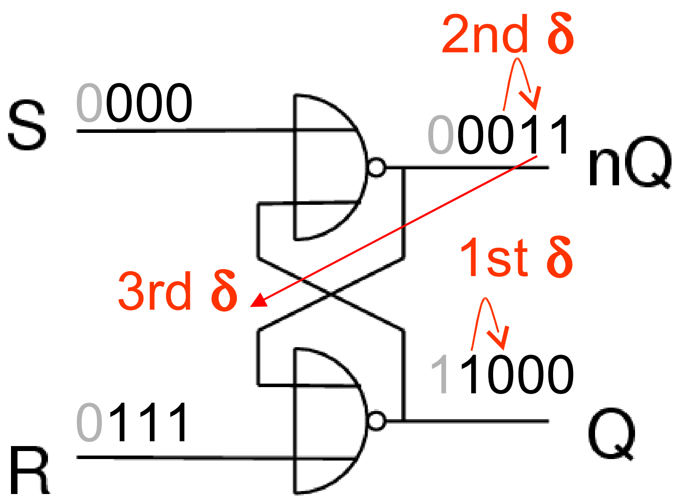
There is an arbitrary number of δ cycles between any 2 physical time instants:




In fact, simulation of delay-less hardware loops might not terminate (don't even advance $T_c$).



# δ-simulation cycles
# Simulation of an RS-Flipflop




```
architecture one
of RS_Flipflop is
begin
process: (R,S,Q,nQ)
    begin
      Q  <= R nor nQ;
      nQ <= S nor Q;
    end process;
end one;
```

| | 0ns | 0ns+δ | 0ns+2δ | 0ns+3δ |
|---|---|---|---|---|
| R | 1 | 1 | 1 | 1 |
| S | 0 | 0 | 0 | 0 |
| Q | 1 | 0 | 0 | 0 |
| nQ | 0 | 0 | 1 | 1 |

**δ cycles reflect the fact that no real gate comes with zero delay.**
☞ should delay-less signal assignments be allowed at all?

# $\delta$-simulation cycles and deterministic simulation semantics

Semantics of
a <= b;
b <= a; ?
Separation into 2 simulation phases results in deterministic semantics
(☞ StateMate).

# VHDL: Evaluation

- Behavioral hierarchy (procedures and functions),
- Structural hierarchy: through structural architectures, but no nested processes,
- No specification of non-functional properties,
- No object-orientation,
- Static number of processes,
- Complicated simulation semantics,
- Too low level for initial specification,
- Good as an intermediate “Esperanto“ or ”assembly” language for hardware generation.

# Summary

VHDL:

- Entities and (behavioral/structural) architectures
- Multiple-valued logic
  - General CSA approach
  - Application to IEEE 1164
- Modeling hardware parallelism by processes
  - Wait statements and sensivity lists
- VHDL semantics: the simulation cycle
  - $\forall$ $\delta$ cycles, deterministic simulation
- Evaluation

# Verilog

HW description language competing with VHDL

Standardized:

- IEEE 1364-1995 (Verilog version 1.0)
- IEEE 1364-2001 (Verilog version 2.0)
- Features similar to VHDL:
- Designs described as connected entities
- Bitvectors and time units are supported
- Features that are different:
- Built-in support for 4-value logic and for logic with 8 strength levels encoded in two bytes per signal.
- More features for transistor-level descriptions
- Less flexible than VHDL.
- More popular in the US (VHDL common in Europe)

# SystemVerilog

Corresponds to Verilog versions 3.0 and 3.1. Includes:

- Additional language elements for modeling behavior
- C data types such as `int`
- Type definition facilities
- Definition of interfaces of HW components as entities
- Mechanism for calling C/C++-functions from Verilog
- Limited mechanism for calling Verilog functions from C.
- Enhanced features for describing the testbench
- Dynamic process creation.
- Interprocess communication and synchronization
- Automatic memory allocation and deallocation.
- Interface for formal verification.

# Using C for ES Design: Motivation

- Many standards (e.g. the GSM and MPEG-standards) are published as C programs
  - ☞ Standards have to be translated if special hardware description languages have to be used
- The functionality of many systems is provided by a mix of hardware and software components
  - ☞ Simulations require an interface between hardware and software simulators unless the same language is used for the description of hardware and software

☞ Attempts to describe software and hardware in the same language. Easier said than implemented.
Various C dialects used for hardware description.

# Drawbacks of a C/C++ Design Flow

- C/C++ is *not* created to design hardware !
- C/C++ does not support
  - Hardware style communication - Signals, protocols
  - Notion of time - Clocks, time sequenced operations
  - Concurrency - Hardware is concurrent, operates in ||
  - Reactivity - Hardware is reactive, responds to stimuli, interacts with its environment (requires handling of exceptions)
  - Hardware data types - Bit type, bit-vector type, multi-valued logic types, signed and unsigned integer types, fixed-point types
- Missing links to hardware during debugging

# SystemC: Required features

Requirements, solutions for modeling HW in a SW language:

- **C++ class library including required functions.**
- **Concurrency:** via processes, controlled by sensivity lists* and calls to wait primitives.
- **Time:** Floating point numbers in SystemC 1.0. Integer values in SystemC 2.0; Includes units such as ps, ns, µs etc*.
- **Support of bit-datatypes**: bitvectors of different lengths; 2- and 4-valued logic; built-in resolution*)
- **Communication:** plug-and-play (pnp) channel model, allowing easy replacement of intellectual property (IP)
- Deterministic behavior not guaranteed.

* Good to know VHDL ☺

# SystemC language architecture

**Channels for MoCs**
Kahn process networks, SDF, etc

**Methodology-specific Channels**
Master/Slave library

**Elementary Channels**
Signal, Timer, Mutex, Semaphore, FIFO, etc

**Core Language**
Module
Ports
Processes
Events
Interfaces
Channels
**Event-driven simulation kernel**

**Data types**
Bits and bit-vectors
Arbitrary precision integers
Fixed-point numbers
4-valued logic types, logic-vectors
C++ user defined types

**C++ Language Standard**

# Summary

- VHDL
- Verilog
- SystemC

Discrete event models